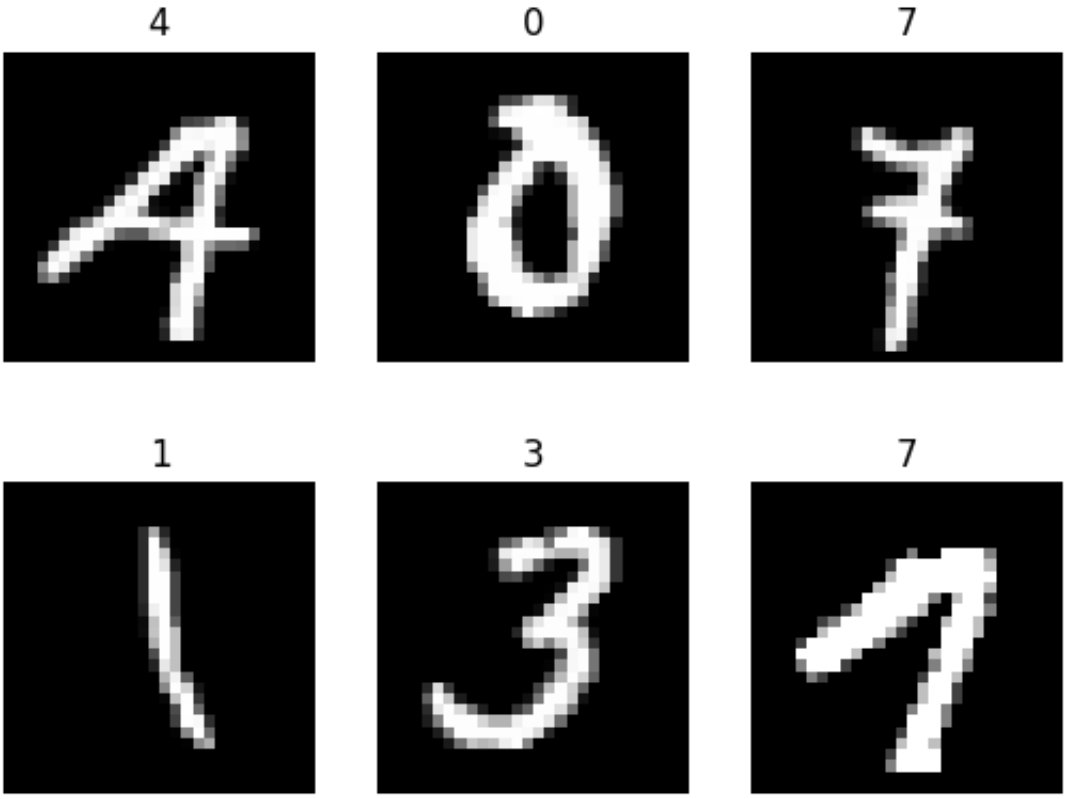
In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Device configuration
# I am using a Mac M3 chip, so I can't use CUDA, but if you are on nvidia, c
device = torch.device("mps")
```

In [8]:
```
# with a large batch size, the optimizer needs a faster learning rate to cor

num_epochs = 5
batch_size = 600
learning_rate = 0.0015
```

Let's load the MNIST data set, first we use torch to install the data, then we use transform to preprocess the images. then using torch utilities we load the data into a dataset, and batch it.

In [3]:
```
# MNIST dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize(
train_dataset = torchvision.datasets.MNIST(root='./data', train=True, downlo
test_dataset = torchvision.datasets.MNIST(root='./data', train=False, downlo

train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=b

# visualize some of the data
examples = iter(train_loader)
example_data, example_targets = next(examples)
for i in range(6):
    plt.subplot(2, 3, i+1)
    plt.title(example_targets[i].item())
    plt.axis('off')
    plt.imshow(example_data[i][0], cmap='gray')
```



Now let's create a simple Convolutional Neural Network,

I choose two Convolutional layers here, to capture basic concepts of the image, The first layer has 16 filters, and for more complex details, the second layer has 32 filters. Max pooling helps make the images resolution smaller, which lowers the dimensionality to ease computations in the next layer. Then ReLU makes the model learn better by avoiding linearity. The fully connected layers at the end are the output nodes and determine which digit is most likely (0-9). This setup works well for MNIST because the images are small and don't have any color data.

In [4]:
```
# make a cnn model with 2 convolutional layers and 2 fully connected layers
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=
        self.fc1 = nn.Linear(32 * 7 * 7, 100)
        self.fc2 = nn.Linear(100, 10)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.pool(self.conv1(x)))
        x = self.relu(self.pool(self.conv2(x)))
```

```
        x = x.view(-1, 32 * 7 * 7)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = CNN().to(device)
```

Here I chose Cross Entropy Loss as the loss function because it is well-suited for multi-class classification problems (like MNIST), It combines both softmax and the negative log likelihood to provide a stable loss calculation that encourages the network to output high probabilities for the right classes.

I chose RAdam with decoupled weight decay here because of this paper https://arxiv.org/abs/1908.03265

In summary,

- regular Adam with decoupled weight decay is an improved version of Adam modifies where the weight decay is performed only after controlling the parameter-wise step size, providing better regularization. This helps prevent overfitting and improves generalization.
- Taking this a step further, we can use RAdam that is different by adapting its learning rate warmup mechanism. This leads to better convergence, especially in the early stages of training.

Then, I chose to exponentially decay the Learning rate so that once the model finishes an epoch, the learning rate isn't as high as the model is already mostly trained.

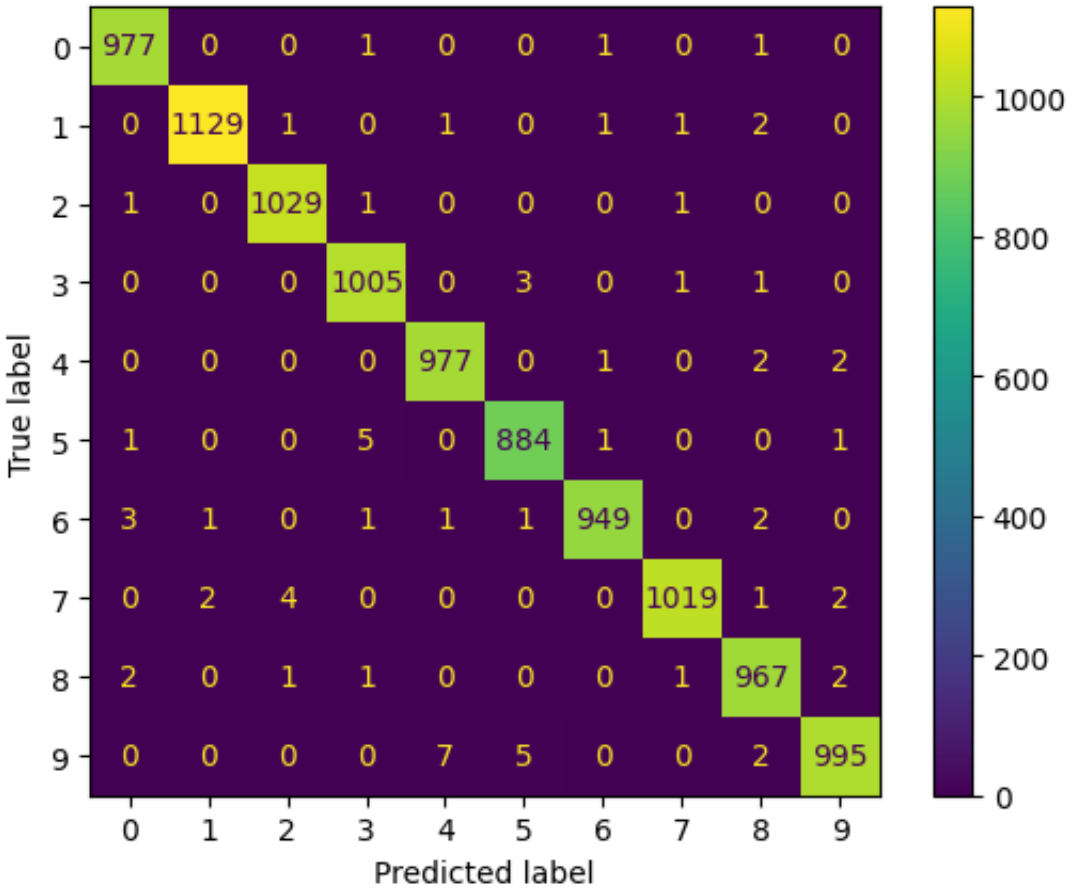
In [24]:
```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.RAdam(model.parameters(), lr=learning_rate, decoupled_weig
scheduler = optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.9)

# Training loop
for epoch in range(num_epochs):
    model.train()
    for i, (images, labels) in enumerate(train_loader):
        images, labels = images.to(device), labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass and optimization
        optimizer.zero_grad()
```

```
        loss.backward()
        optimizer.step()

        if (i + 1) % (len(train_loader)//4) == 0:
            print(f'Epoch [{epoch + 1}/{num_epochs}], Step [{i + 1}/{len(tra
    scheduler.step()
```

```
Epoch [1/5], Step [25/100], Loss: 0.000024
Epoch [1/5], Step [50/100], Loss: 0.000017
Epoch [1/5], Step [75/100], Loss: 0.000075
Epoch [1/5], Step [100/100], Loss: 0.000131
Epoch [2/5], Step [25/100], Loss: 0.000007
Epoch [2/5], Step [50/100], Loss: 0.000013
Epoch [2/5], Step [75/100], Loss: 0.000006
Epoch [2/5], Step [100/100], Loss: 0.000017
Epoch [3/5], Step [25/100], Loss: 0.000003
Epoch [3/5], Step [50/100], Loss: 0.000003
Epoch [3/5], Step [75/100], Loss: 0.000005
Epoch [3/5], Step [100/100], Loss: 0.000002
Epoch [4/5], Step [25/100], Loss: 0.000005
Epoch [4/5], Step [50/100], Loss: 0.000003
Epoch [4/5], Step [75/100], Loss: 0.000002
Epoch [4/5], Step [100/100], Loss: 0.000002
Epoch [5/5], Step [25/100], Loss: 0.000003
Epoch [5/5], Step [50/100], Loss: 0.000003
Epoch [5/5], Step [75/100], Loss: 0.000006
Epoch [5/5], Step [100/100], Loss: 0.000099
```

In [25]:
```
# Evaluation and confusion matrix

model.eval()
y_pred = []
y_true = []

with torch.no_grad():
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        y_true.extend(labels.cpu().numpy())
        y_pred.extend(predicted.cpu().numpy())

# Confusion matrix
cm = confusion_matrix(y_true, y_pred)
cm_display = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=np.a
cm_display.plot()
plt.show()

# Model summary of accuracy and hyperparameters
```

```
print("Model Architecture:")
print(model)
print("\nModel Evaluation:")
print(f"Accuracy: {np.mean(np.array(y_true) == np.array(y_pred))}")
print("\nHyperparameters:")
print(f"Learning Rate: {learning_rate}")
print(f"Batch Size: {batch_size}")
print(f"Number of Epochs: {num_epochs}")
```



```
Model Architecture:
CNN(
  (conv1): Conv2d(1, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (conv2): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (fc1): Linear(in_features=1568, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=10, bias=True)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mod
e=False)
  (relu): ReLU()
)

Model Evaluation:
Accuracy: 0.9931

Hyperparameters:
Learning Rate: 0.0015
Batch Size: 600
Number of Epochs: 5
```

Using the above parameters, I can get a pretty good score (roughly 99.3%) given the histogram of Kaggle scores. In this case, my model seems to be on the higher end of pure CNNs, even though my architecture is pretty simple. I believe that has to do with the optimizers and the learning rate decay, as that helped my model break throguh from 98%.

Adding a data augmentation step would surely increase my accuracy, perhaps a dropout layer as well to improve regularization, and creating a more complex CNN could also be helpful. But based on the Kaggle score, I probably have reached an inherit limit of pure CNNs.

According to the paper that currently holds the highest score, https://arxiv.org/pdf/2001.09136v6, using Homogeneous Vector Capsules (HVCs), their simple convolutional network achieves state-of-the-art MNIST classification with fewer parameters, fewer epochs, and without a routing mechanism. Unlike regular neurons, which output a scalar value, capsules output a vector, which contain more information about the properties of a detected entity (like, for example, their presence, orientation, maybe their scale).